**Project Title: Measure Energy Consumption**

**Project Overview:**

* Provide an introduction to the project, explaining its purpose, significance, and the problem it aims to address. Include a brief description of what the project entails.
* **Project Goals**:
* Clearly state the objectives and goals of the project. What do you hope to achieve by measuring energy consumption?

**Design Thinking Process**:

* Explain how the design thinking methodology will be applied to this project. Highlight its importance in problem-solving and innovation.

**Empathize**:

* Describe the process of understanding the needs and concerns of users or stakeholders related to energy consumption. Include methods used for data collection and user feedback.
* **Define**:
* Define the problem statement based on the insights gathered during the empathize phase. Clearly articulate the challenges or issues related to energy consumption that the project aims to address.

**Ideate**:

* Discuss the creative brainstorming and idea generation process. Explain how potential solutions or approaches to measuring energy consumption were generated.

**Prototype**:

* Describe the development of prototypes or mock-ups for the energy consumption measurement system. Include details about the design, features, and functionality of the prototypes.

**Test**:

* Explain the testing phase, where the prototypes or concepts were evaluated. Discuss the criteria for testing and any adjustments or improvements made based on feedback.

**Implement**:

* Detail the implementation of the final solution for measuring energy consumption. Include information about hardware, software, and any other components used.

**Monitor and Iterate**:

* Discuss how the project will be monitored after implementation to ensure its effectiveness. Explain the iterative process for making improvements based on ongoing feedback and data.

**Key Components**:

* List and describe the key components of the energy consumption measurement system, including sensors, data collection methods, and reporting mechanisms.

**Project Timeline**:

* Provide a timeline or schedule that outlines the key milestones and phases of the project, including start and end dates for each phase.

# **Importing packages**

In [1]:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib as mat

import statsmodels.api as sm

from fbprophet import Prophet

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error

from statsmodels.tsa.stattools import adfuller

In [2]:

pd.plotting.register\_matplotlib\_converters()

In [3]:

mat.rcParams.update({'figure.figsize':(20,15), 'font.size': 14})

# **Reading the data**

In [4]:

energy\_consumption = pd.read\_csv('../input/hourly-energy-consumption/PJME\_hourly.csv')

# **Preprocessing**

In [5]:

energy\_consumption.head()

Out[5]:

|  | Datetime | PJME\_MW |
| --- | --- | --- |
| 0 | 2002-12-31 01:00:00 | 26498.0 |
| 1 | 2002-12-31 02:00:00 | 25147.0 |
| 2 | 2002-12-31 03:00:00 | 24574.0 |
| 3 | 2002-12-31 04:00:00 | 24393.0 |
| 4 | 2002-12-31 05:00:00 | 24860.0 |

In [6]:

energy\_consumption.dtypes

Out[6]:

Datetime object

PJME\_MW float64

dtype: object

In [7]:

energy\_consumption.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 145366 entries, 0 to 145365

Data columns (total 2 columns):

Datetime 145366 non-null object

PJME\_MW 145366 non-null float64

dtypes: float64(1), object(1)

memory usage: 2.2+ MB

In [8]:

energy\_consumption['Datetime'] = pd.to\_datetime(energy\_consumption['Datetime'])

In [9]:

energy\_consumption = energy\_consumption.set\_index('Datetime').resample('H').sum()

In [10]:

energy\_consumption.info()

<class 'pandas.core.frame.DataFrame'>

DatetimeIndex: 145392 entries, 2002-01-01 01:00:00 to 2018-08-03 00:00:00

Freq: H

Data columns (total 1 columns):

PJME\_MW 145392 non-null float64

dtypes: float64(1)

memory usage: 2.2 MB

In [11]:

hours\_no\_consumption = energy\_consumption.loc[energy\_consumption['PJME\_MW'] == 0].copy()

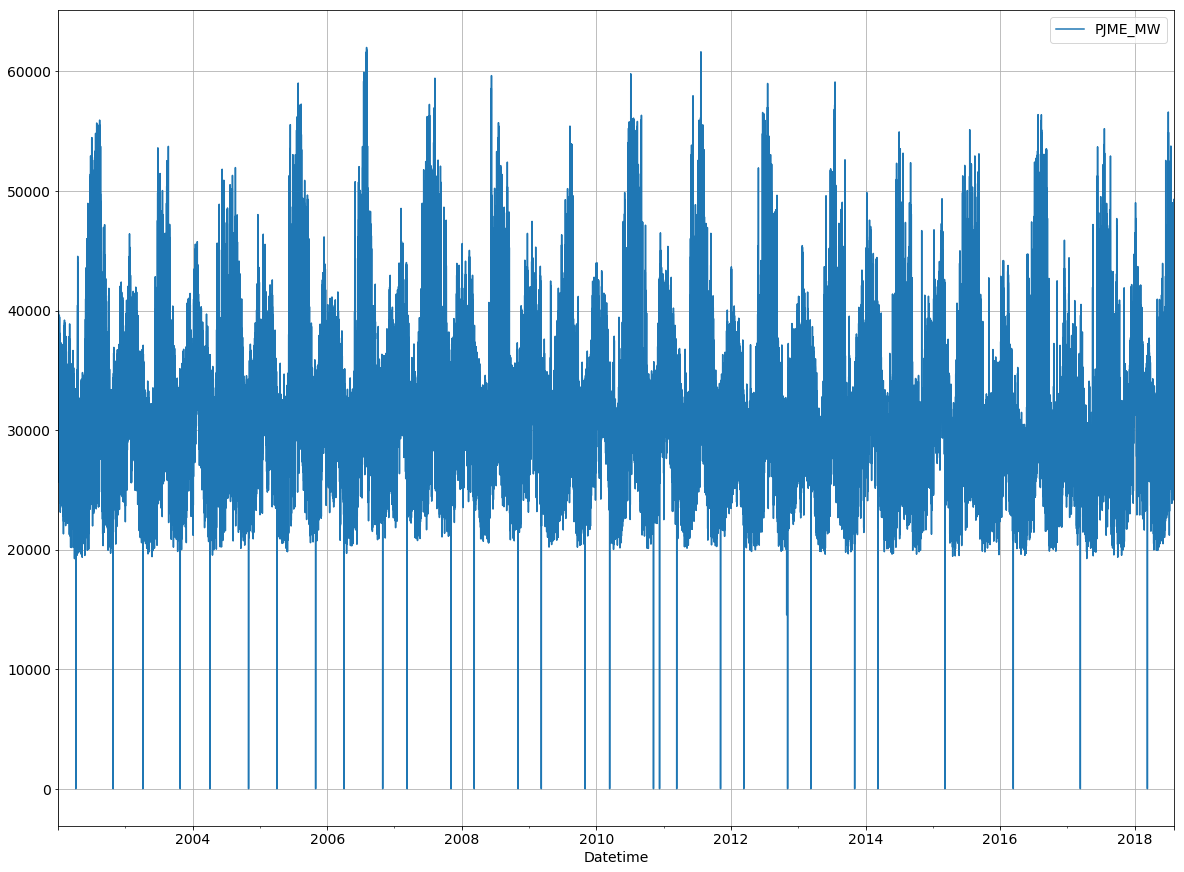
# **Exploratory Analysis**

In [12]:

energy\_consumption.plot(grid=True)

*# plt.yscale('log')*

plt.show()



In [13]:

energy\_consumption.describe()

Out[13]:

|  | PJME\_MW |
| --- | --- |
| count | 145392.000000 |
| mean | 32074.486024 |
| std | 6479.660890 |
| min | 0.000000 |
| 25% | 27571.000000 |
| 50% | 31420.000000 |
| 75% | 35648.250000 |
| max | 62009.000000 |

* Total energy consumption falls between (0 - 62009) MW
* Averge energy consumption per hour is 32074.5 MW with std of 6479.7

In [14]:

energy\_consumption.loc[~energy\_consumption.index.isin(hours\_no\_consumption.index)].describe()

Out[14]:

|  | PJME\_MW |
| --- | --- |
| count | 145362.000000 |
| mean | 32081.105598 |
| std | 6463.923399 |
| min | 14544.000000 |
| 25% | 27574.000000 |
| 50% | 31421.000000 |
| 75% | 35650.750000 |
| max | 62009.000000 |

Removing the hours with no energy consumption data:

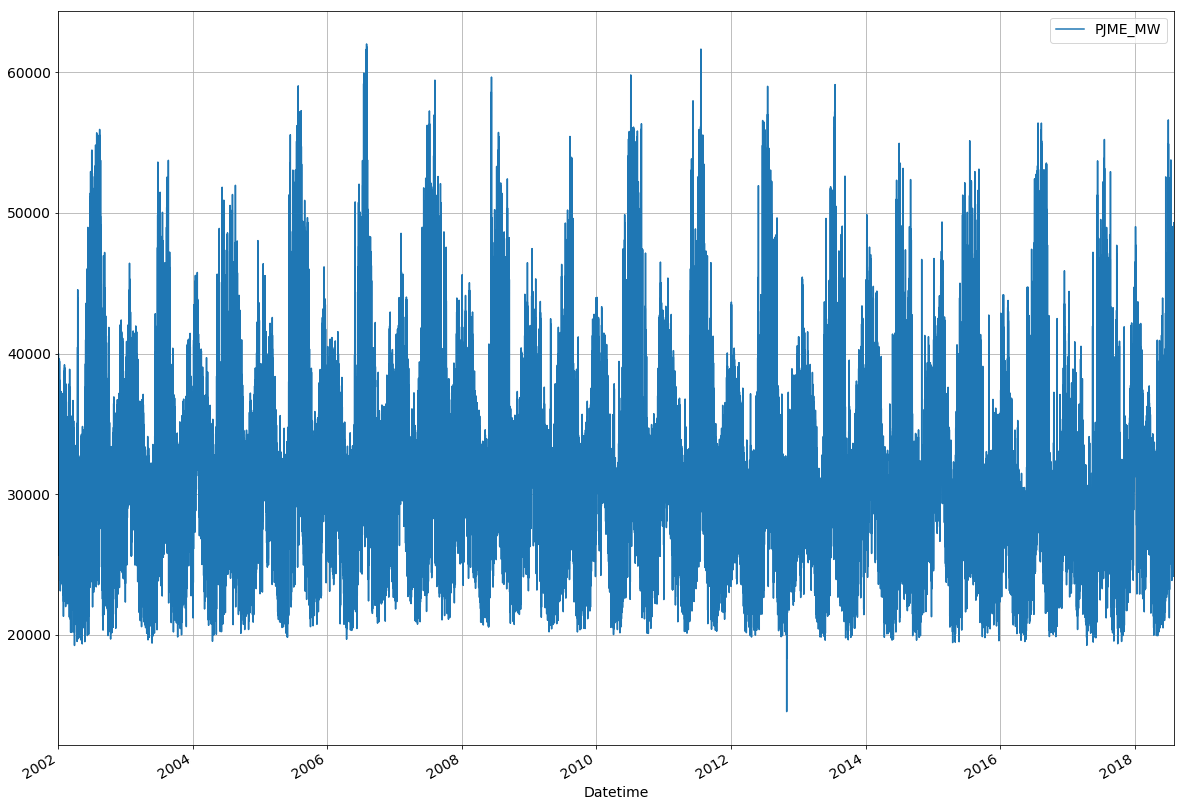
* The average energy consumption didn't change alot.
* Minimum recorded energy consumtion is 14544 MW.

In [15]:

energy\_consumption.loc[~energy\_consumption.index.isin(hours\_no\_consumption.index)].plot(grid=True)

*# plt.yscale('log')*

plt.show()



95% of the energy consumption falls between:

In [16]:

energy\_consumption['PJME\_MW'].quantile([0.025]).values[0]

Out[16]:

21678.0

In [17]:

energy\_consumption['PJME\_MW'].quantile([0.975]).values[0]

Out[17]:

47523.225000000006

In [18]:

energy\_consumption.loc[(energy\_consumption['PJME\_MW'] >= energy\_consumption['PJME\_MW'].quantile([0.025]).values[0])

&

(energy\_consumption['PJME\_MW'] <= energy\_consumption['PJME\_MW'].quantile([0.975]).values[0])].describe()

Out[18]:

|  | PJME\_MW |
| --- | --- |
| count | 138123.000000 |
| mean | 31872.380813 |
| std | 5575.156271 |
| min | 21678.000000 |
| 25% | 27801.500000 |
| 50% | 31420.000000 |
| 75% | 35363.000000 |
| max | 47523.000000 |

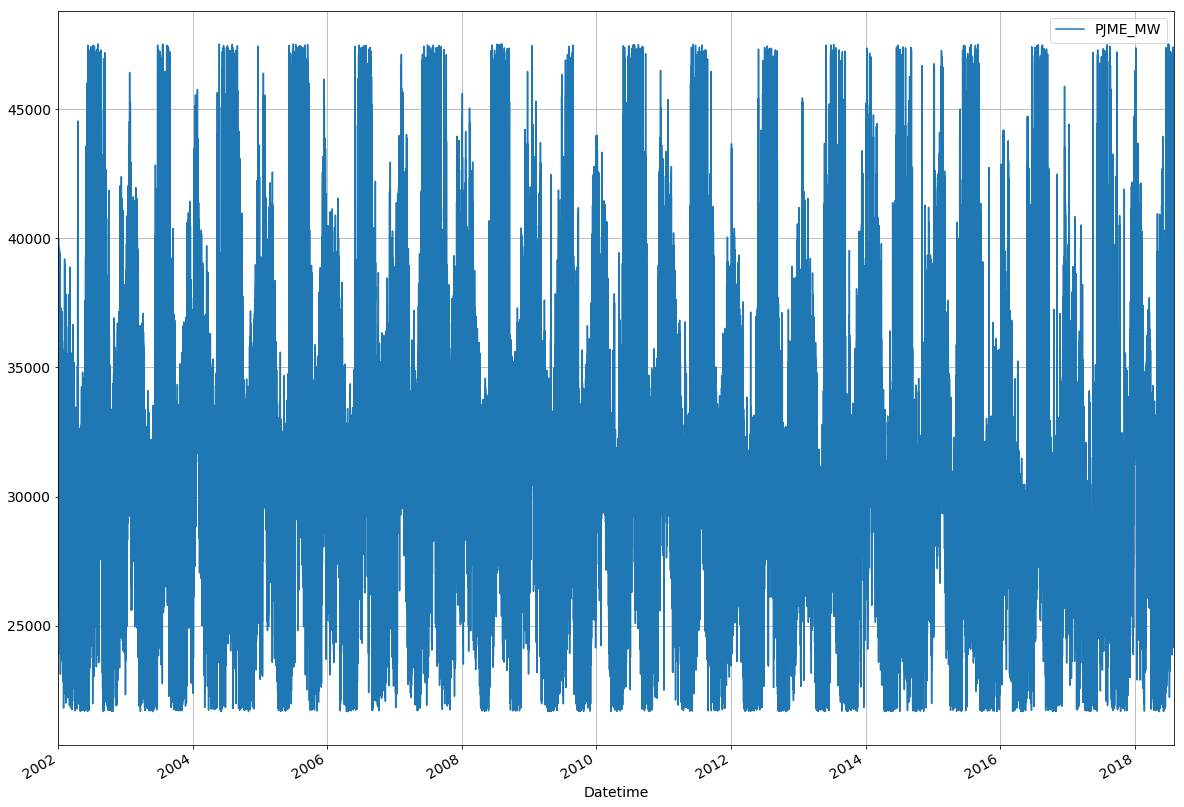
In [19]:

energy\_consumption.loc[(energy\_consumption['PJME\_MW'] >= energy\_consumption['PJME\_MW'].quantile([0.025]).values[0])

&

(energy\_consumption['PJME\_MW'] <= energy\_consumption['PJME\_MW'].quantile([0.975]).values[0])].plot(grid=True)

plt.show()



In [20]:

energy\_consumption.resample('YS').mean().sort\_values('PJME\_MW',ascending=False).head(1)

Out[20]:

|  | PJME\_MW |
| --- | --- |
| Datetime |  |
| 2007-01-01 | 33605.794292 |

In [21]:

energy\_consumption.resample('YS').mean().sort\_values('PJME\_MW',ascending=False).tail(1)

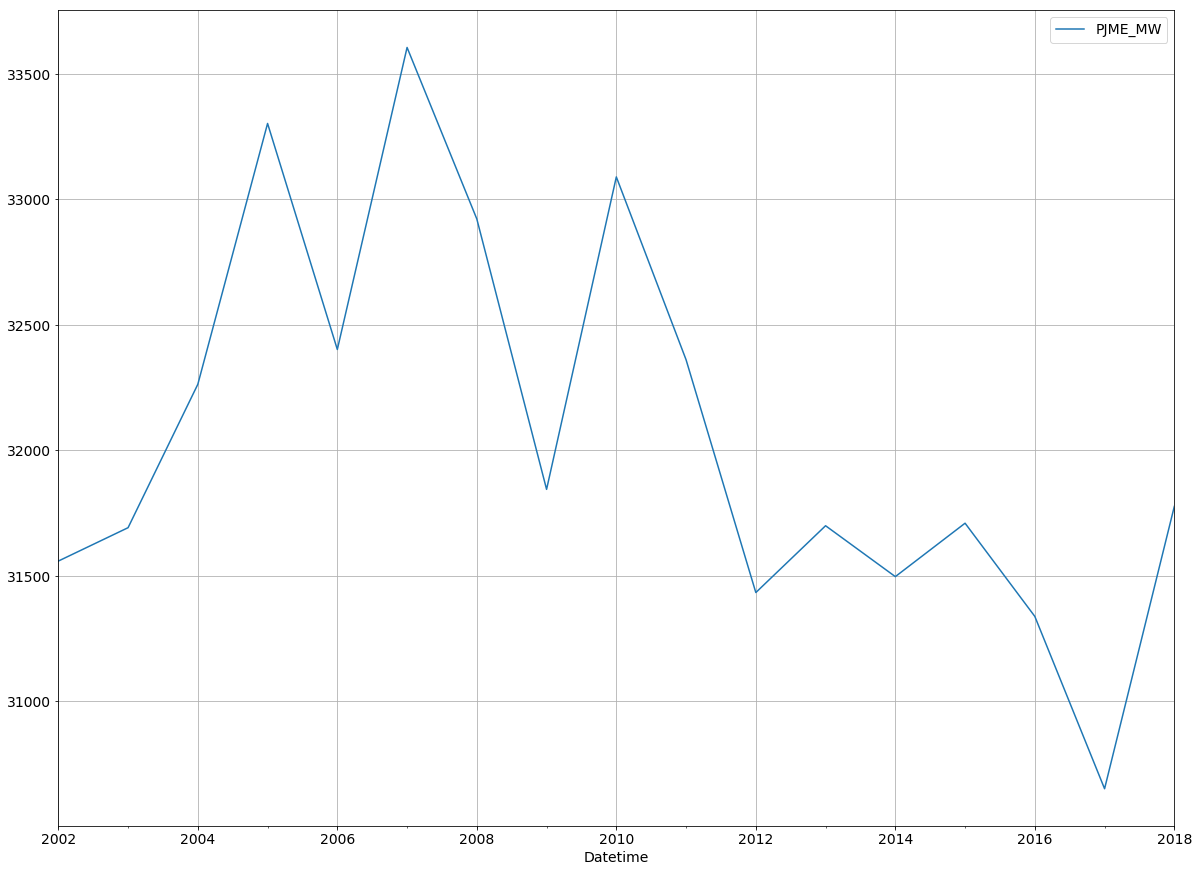
Out[21]:

|  | PJME\_MW |
| --- | --- |
| Datetime |  |
| 2017-01-01 | 30650.911644 |

In [22]:

energy\_consumption.resample('YS')[['PJME\_MW']].mean().plot(grid=True)

plt.show()

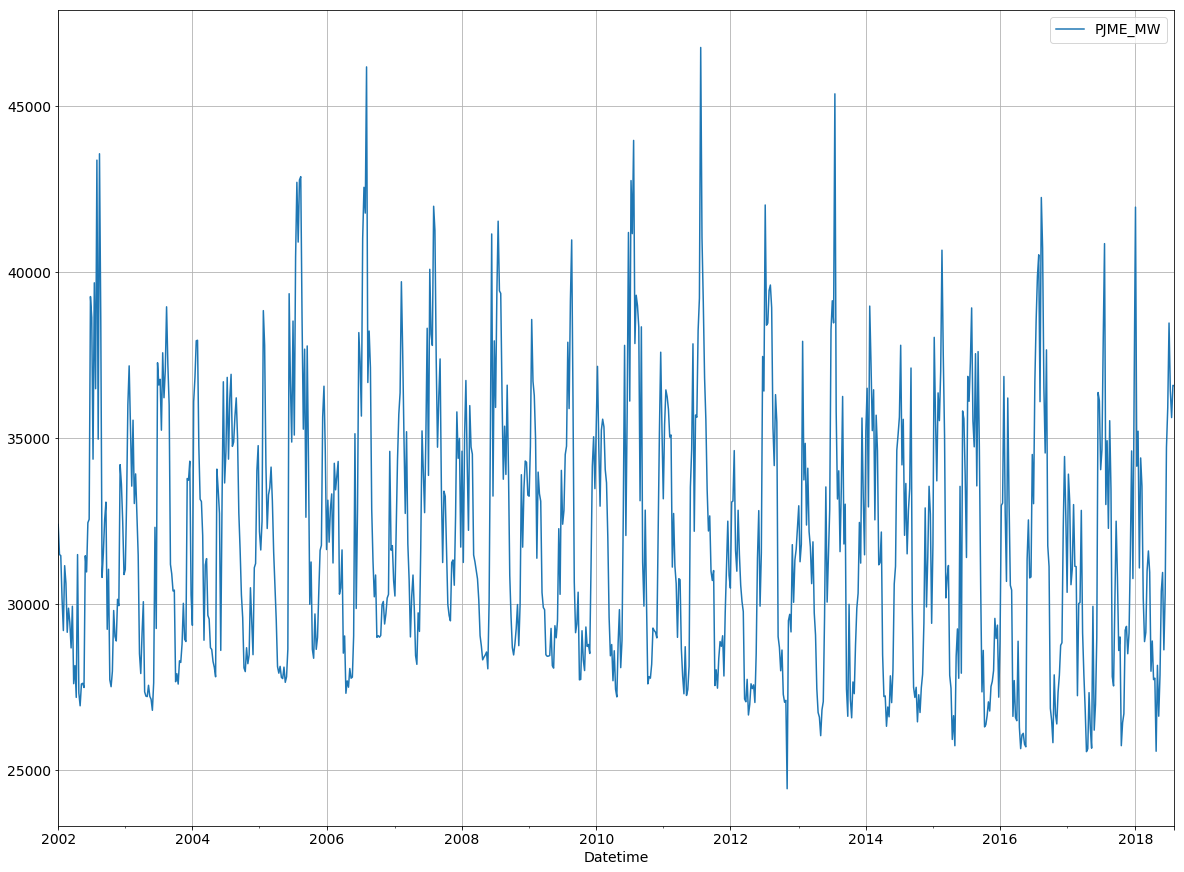


* 2007 Had the highest average energy consumption per hour, 2005 follows
* 2017 has the lowest, interesting!

In [23]:

energy\_consumption.resample('W').mean().plot(grid=True)

plt.show()



In [24]:

energy\_consumption.loc['01-01-2018':].resample('W').mean().sort\_values('PJME\_MW',ascending=False).head(1)

Out[24]:

|  | PJME\_MW |
| --- | --- |
| Datetime |  |
| 2018-01-07 | 41951.339286 |

* first week in Jan 2018 had the highest energy consumption

In [25]:

energy\_consumption['Hour'] = energy\_consumption.index.hour

In [26]:

max\_hour = energy\_consumption.loc[energy\_consumption.loc['06-01-2018':'07-31-2018'].resample('D')['PJME\_MW'].idxmax()].copy()

* The highest energy consumption in 2018 was in 2018-07-03 "Tuesday"

In [27]:

df = energy\_consumption.loc['06-01-2018':'07-31-2018'].resample('D')[['PJME\_MW']].sum()

\_ = plt.plot(df.index, df.PJME\_MW)

for H **in** max\_hour['Hour'].unique():

df = energy\_consumption.loc['06-01-2018':'07-31-2018'].resample('D')[['PJME\_MW']].sum()

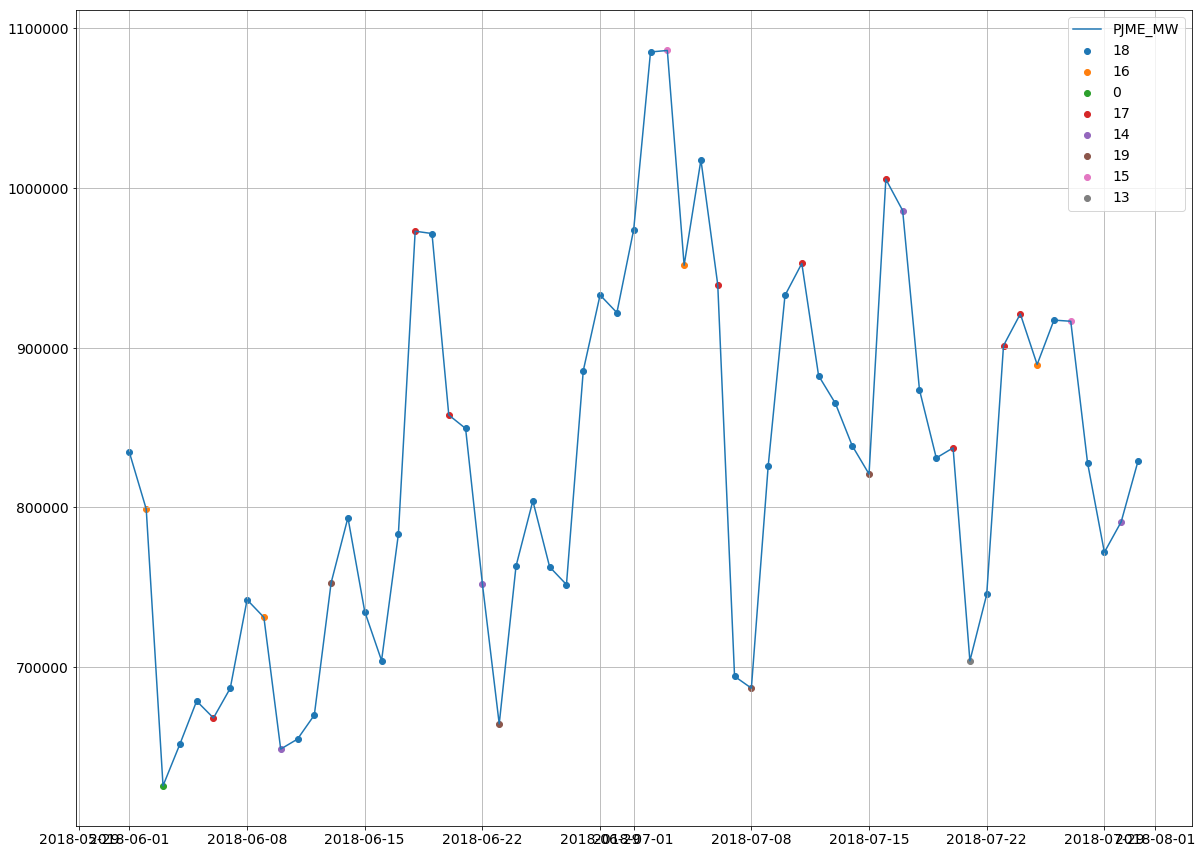
df = df.loc[max\_hour.loc[max\_hour['Hour'] == H].index.date].copy()

\_ = plt.scatter( x = df.index, y = df.PJME\_MW, label = H)

plt.grid()

plt.legend()

plt.show()



* 6 PM is the hour with the highest consumtion in most of the days for the selected period

In [28]:

dayofweek = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']

energy\_consumption['Day of Week'] = energy\_consumption.index.dayofweek

energy\_consumption['Day of Week'] = energy\_consumption['Day of Week'].apply(lambda x: dayofweek[x])

In [29]:

energy\_consumption.loc['01-01-2018':].groupby([energy\_consumption.loc['01-01-2018':].index.date,

'Day of Week'])[['PJME\_MW']].sum().sort\_values('PJME\_MW',

ascending=False).head()

Out[29]:

|  |  | PJME\_MW |
| --- | --- | --- |
|  | Day of Week |  |
| 2018-07-03 | Tuesday | 1086193.0 |
| 2018-07-02 | Monday | 1085235.0 |
| 2018-01-05 | Friday | 1060747.0 |
| 2018-01-06 | Saturday | 1045578.0 |
| 2018-07-05 | Thursday | 1017657.0 |

* The highest energy consumption in 2018 was in 2018-07-03 "Tuesday"

In [30]:

df = energy\_consumption.loc['06-01-2018':'07-31-2018'].resample('D')[['PJME\_MW']].sum()

\_ = plt.plot(df.index, df.PJME\_MW)

for D **in** energy\_consumption['Day of Week'].unique():

df = energy\_consumption.loc['06-01-2018':'07-31-2018'].loc[energy\_consumption['Day of Week'] == D].copy()

df = df.resample('D')[['PJME\_MW']].sum().copy()

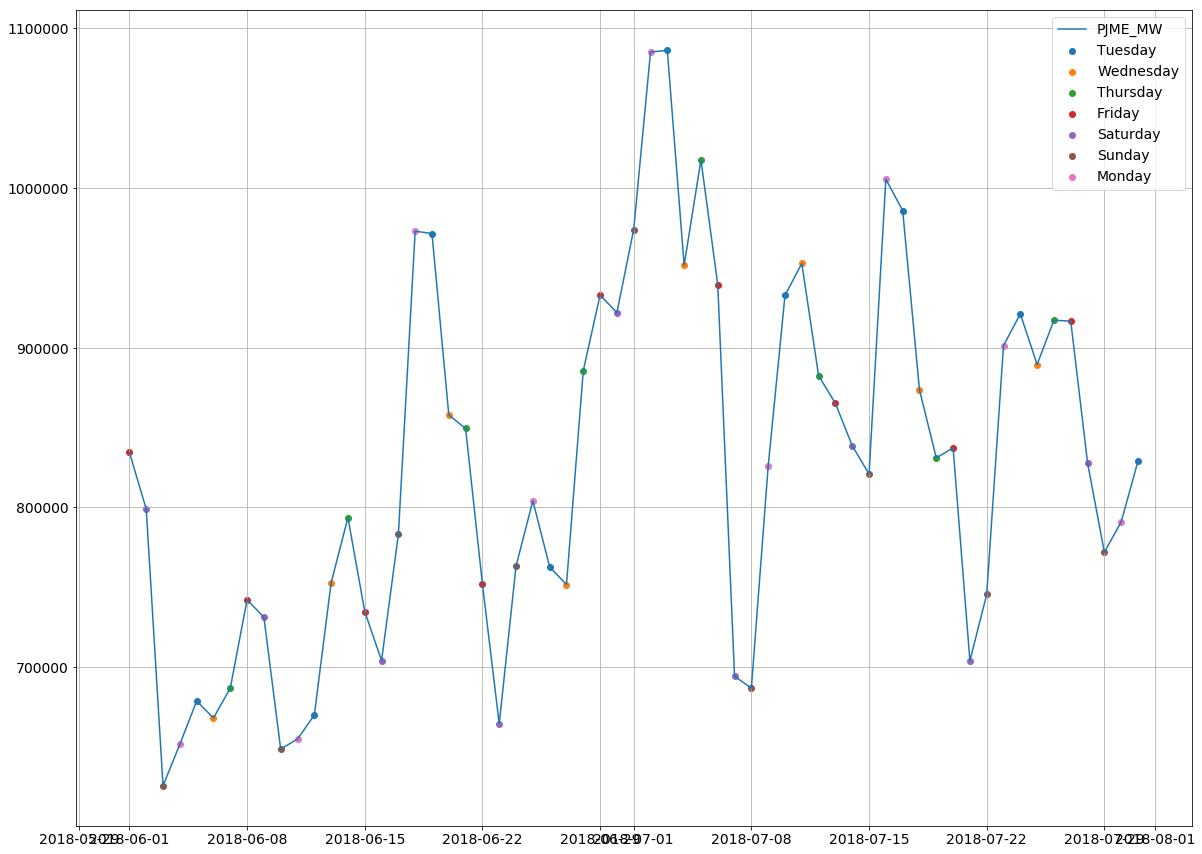
df = df.loc[df['PJME\_MW'] != 0].copy()

\_ = plt.scatter( x = df.index, y = df.PJME\_MW, label = D)

plt.grid()

plt.legend()

plt.show()



* peaks dominated by Mondays and Tuesdays
* whereas, troughs are dominated by Saturdays and Sundays

In [31]:

energy\_consumption.loc['01-01-2017':'31-12-2017'].resample('Q').sum().sort\_values('PJME\_MW',ascending=False)

Out[31]:

|  | PJME\_MW | Hour |
| --- | --- | --- |
| Datetime |  |  |
| 2017-09-30 | 73627696.0 | 25392 |
| 2017-03-31 | 66768754.0 | 24840 |
| 2017-12-31 | 65425156.0 | 25392 |
| 2017-06-30 | 62680380.0 | 25116 |

* The highest energy consumption Qurter in 2017 was in third Qurter (Summer).

In [32]:

year\_quarter\_con = energy\_consumption.copy()

year\_quarter\_con['Year'] = year\_quarter\_con.index.year

year\_quarter\_con['Quarter'] = year\_quarter\_con.index.quarter

In [33]:

year\_quarter\_con = year\_quarter\_con.groupby(['Year','Quarter'])[['PJME\_MW']].sum()

year\_quarter\_con.iloc[year\_quarter\_con.reset\_index().groupby(['Year'])['PJME\_MW'].idxmax()]

Out[33]:

|  |  | PJME\_MW |
| --- | --- | --- |
| Year | Quarter |  |
| 2002 | 3 | 79108260.0 |
| 2003 | 3 | 76773824.0 |
| 2004 | 3 | 76182651.0 |
| 2005 | 3 | 83771227.0 |
| 2006 | 3 | 80581425.0 |
| 2007 | 3 | 80811919.0 |
| 2008 | 3 | 78821422.0 |
| 2009 | 3 | 74989501.0 |
| 2010 | 3 | 81481624.0 |
| 2011 | 3 | 79785091.0 |
| 2012 | 3 | 78837294.0 |
| 2013 | 3 | 75607834.0 |
| 2014 | 1 | 74726381.0 |
| 2015 | 3 | 76681092.0 |
| 2016 | 3 | 80488648.0 |
| 2017 | 3 | 73627696.0 |
| 2018 | 1 | 69982662.0 |

In [34]:

energy\_consumption['Quarter'] = energy\_consumption.index.quarter

energy\_consumption['Month'] = energy\_consumption.index.month

In [35]:

df = energy\_consumption.loc[:'31-07-2018'].resample('MS')[['PJME\_MW']].sum()

\_ = plt.plot( df.index, df.PJME\_MW)

for Q **in** energy\_consumption['Quarter'].unique():

df = energy\_consumption.loc[:'31-07-2018'].loc[energy\_consumption['Quarter'] == Q].resample('MS')[['PJME\_MW']].sum().copy()

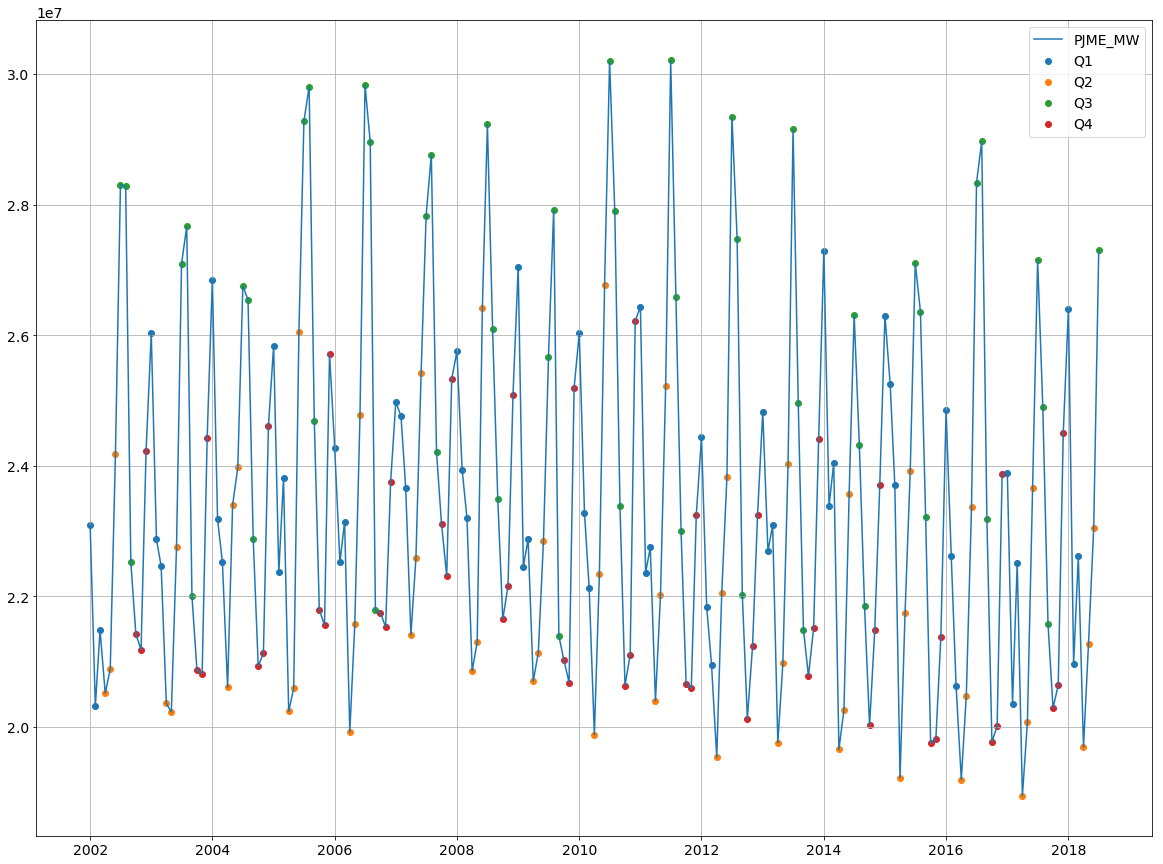
df = df.loc[df['PJME\_MW'] != 0].copy()

\_ = plt.scatter( x = df.index, y = df.PJME\_MW, label = 'Q' +str(Q))

plt.grid()

plt.legend()

plt.show()



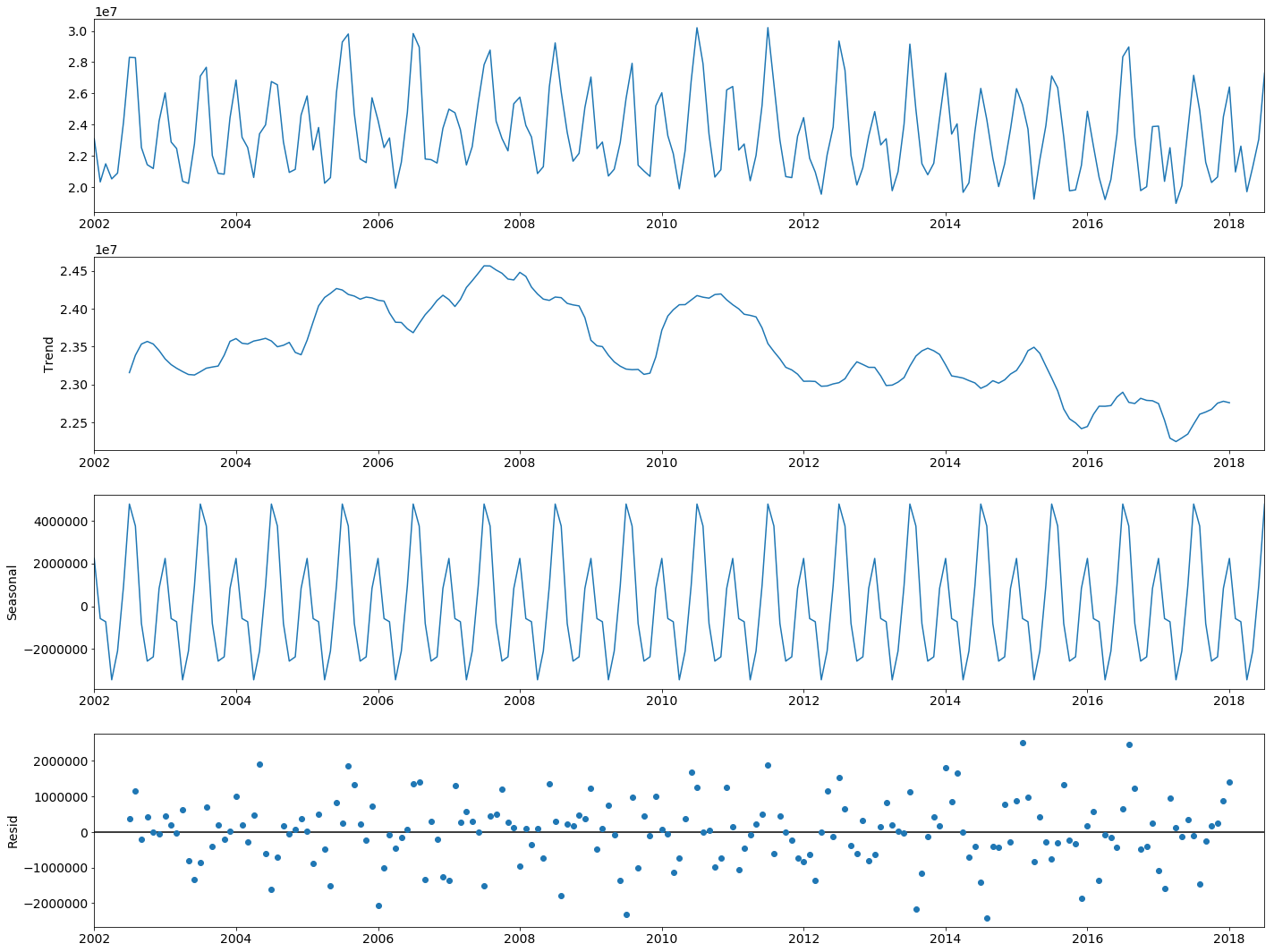
* okay so summer had always been the season with the highest consumption through the years.
* 2nd and 4th quarters had the lowest consumption

In [36]:

decompose = sm.tsa.seasonal\_decompose(energy\_consumption.loc[:'31-07-2018'].resample('MS')[['PJME\_MW']].sum())

decompose.plot()

plt.show()



In [37]:

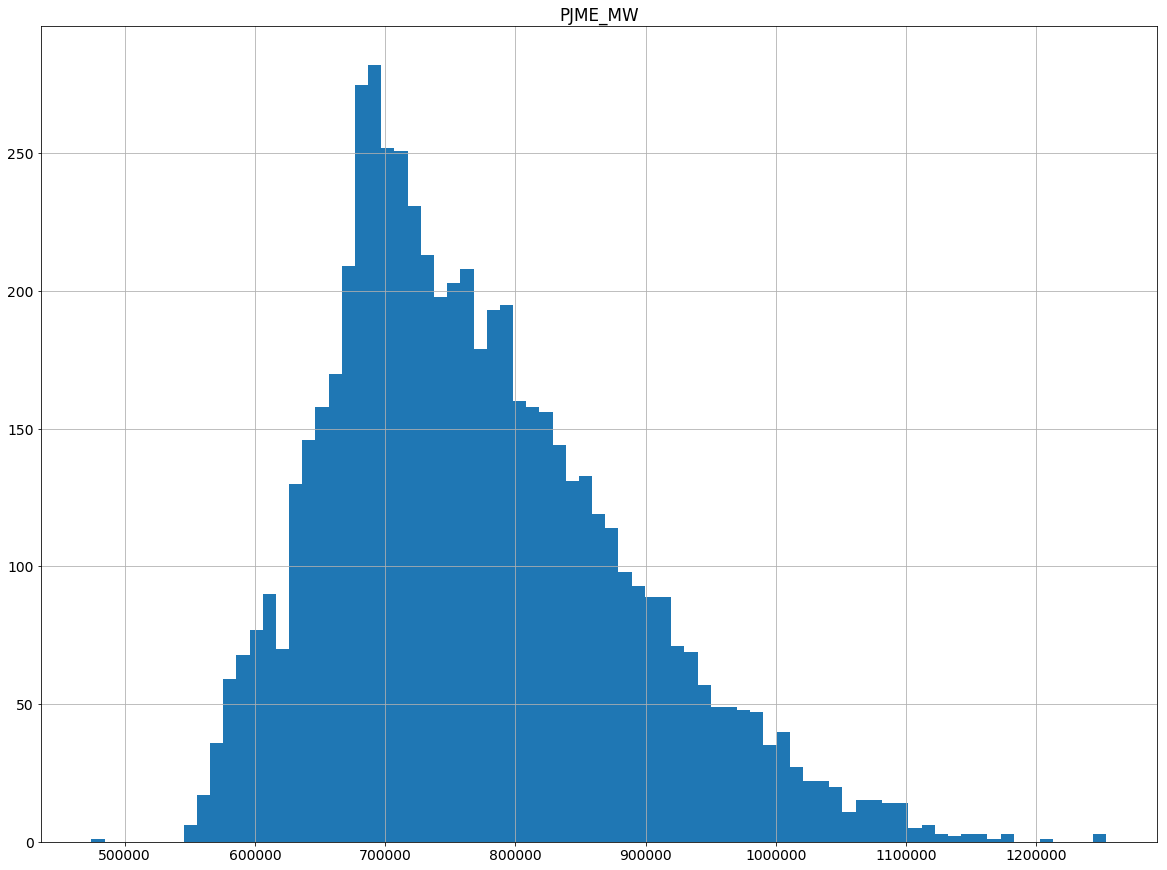
day\_consum = energy\_consumption.loc[:'31-07-2018'].resample('D')[['PJME\_MW']].sum()

day\_consum.hist(bins=int(np.sqrt(len(day\_consum))))

Out[37]:

array([[<matplotlib.axes.\_subplots.AxesSubplot object at 0x7f9d724e3a90>]],

dtype=object)



In [38]:

day\_consum.reset\_index(inplace=True)

In [39]:

day\_consum['Datetime'] = pd.to\_datetime(day\_consum['Datetime'])

In [40]:

day\_consum.index = pd.DatetimeIndex(day\_consum['Datetime'],freq='D')

In [41]:

day\_consum.drop(['Datetime'],1,inplace=True)

# **Stationarity Test**

In [42]:

result = adfuller(day\_consum['PJME\_MW'].values)

print('ADF Statistic: **%f**' % result[0])

print('p-value: **%f**' % result[1])

print('Critical Values:')

for key, value **in** result[4].items():

print('**\t%s**: **%.3f**' % (key, value))

if result[0] <= value:

print('Stationary at ' + key)

else:

print('Non-Stationary at ' + key)

ADF Statistic: -8.276309

p-value: 0.000000

Critical Values:

1%: -3.431

Stationary at 1%

5%: -2.862

Stationary at 5%

10%: -2.567

Stationary at 10%

In [43]:

day\_consum['ds'] = day\_consum.index

day\_consum.rename(columns={'PJME\_MW':'y'},inplace=True)

# **Train Test Data**

In [44]:

split\_date = '06-30-2018'

train = day\_consum.loc[:split\_date].copy()

test = day\_consum.loc[split\_date:].copy()

# **Time Series Prediction**

In [45]:

model = Prophet()

In [46]:

model.fit(train)

Out[46]:

<fbprophet.forecaster.Prophet at 0x7f9d7248a748>

In [47]:

future = model.make\_future\_dataframe(periods=len(test))

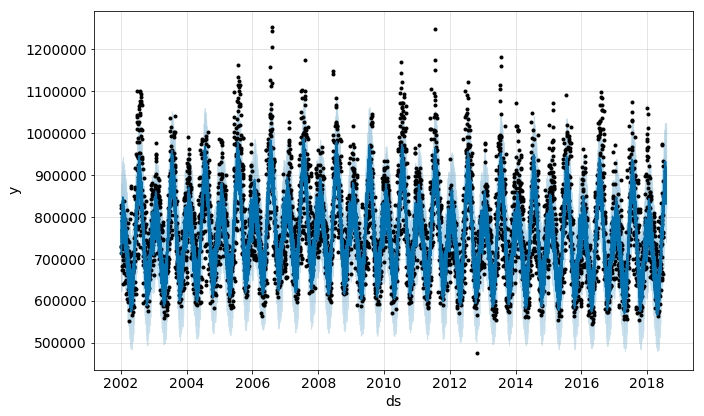
In [48]:

forecast = model.predict(future)

In [49]:

model.plot(forecast)

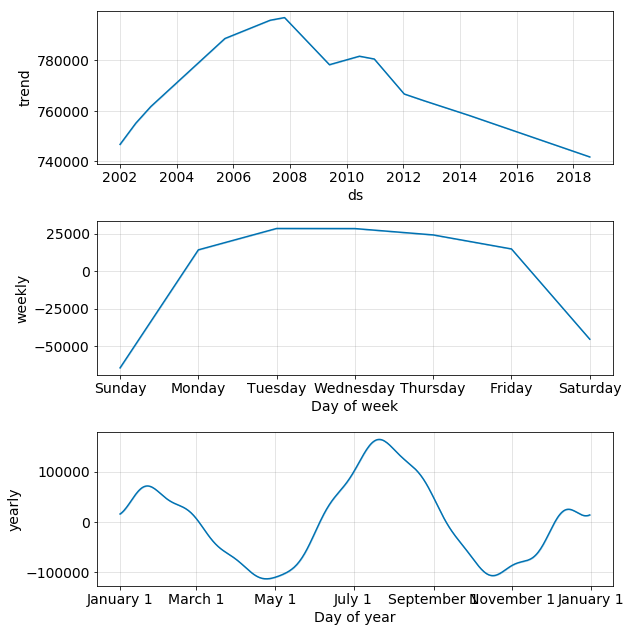
plt.show()



In [50]:

model.plot\_components(forecast)

plt.show()



In [51]:

prediction\_vs\_real = forecast.set\_index('ds')[['yhat', 'yhat\_lower', 'yhat\_upper']].join(day\_consum.set\_index('ds'))

In [52]:

def calculate\_forecast\_errors(df, prediction\_size):

*"""Calculate MAPE and MAE of the forecast.*

*Args:*

*df: joined dataset with 'y' and 'yhat' columns.*

*prediction\_size: number of days at the end to predict.*

*"""*

*# Make a copy*

df = df.copy()

*# Now we calculate the values of e\_i and p\_i according to the formulas given in the article above.*

df['e'] = df['y'] - df['yhat']

df['p'] = 100 \* df['e'] / df['y']

*# Recall that we held out the values of the last `prediction\_size` days*

*# in order to predict them and measure the quality of the model.*

*# Now cut out the part of the data which we made our prediction for.*

predicted\_part = df[-prediction\_size:]

*# Define the function that averages absolute error values over the predicted part.*

error\_mean = lambda error\_name: np.mean(np.abs(predicted\_part[error\_name]))

*# Now we can calculate MAPE and MAE and return the resulting dictionary of errors.*

return {'MAPE': error\_mean('p'), 'MAE': error\_mean('e')}

In [53]:

calculate\_forecast\_errors(prediction\_vs\_real, len(test))

Out[53]:

{'MAPE': 8.783032537399167, 'MAE': 76745.33751399578}

**MEASURE ENERGY CONSUMPTION PREPROCESSING IN DATASET**

[TensorFlow](https://tensorflow.org/) and [Neural Designer](https://www.neuraldesigner.com/) are popular machine learning platforms developed by [Google](https://research.google/teams/brain/) and [Artelnics](https://www.artelnics.com/), respectively.

Although all those frameworks are based on neural networks, they present essential differences in functionality, usability, performance, consumption, etc.

This post compares the energy consumption of TensorFlow and Neural Designer using the GPU for an approximation benchmark.

As we will see, Neural Designer consumes **42**% less than its competitor machine learning platform.

In this article, we provide all the steps you need to reproduce the result using the [free trial](https://www.neuraldesigner.com/free-trial) of Neural Designer.

**Contents:**

* [Introduction](https://www.neuraldesigner.com/blog/energy-consumption-comparison/#Introduction).
* [Benchmark application](https://www.neuraldesigner.com/blog/energy-consumption-comparison/#BenchmarkApplication).
* [Reference computer](https://www.neuraldesigner.com/blog/energy-consumption-comparison/#ReferenceComputer).
* [Reference electricity consumption meter](https://www.neuraldesigner.com/blog/energy-consumption-comparison/#Referenceelectricityconsumptionmeter).
* [Results](https://www.neuraldesigner.com/blog/energy-consumption-comparison/#Results).
* [Conclusions](https://www.neuraldesigner.com/blog/energy-consumption-comparison/#Conclusions).

## **Introduction**

Two of the most essential features of machine learning platforms are their training speed and the total amount of energy consumed during this process.

In most cases, modeling huge data sets is very expensive in computational terms, which leads to a high economic cost of neural network training and a high environmental impact.

Thus, this article aims to measure the GPU energy consumption of TensorFlow and Neural Designer for a benchmark application. Also, a couple of instructions are given to enable anyone to repeat this one or a similar benchmark and check on their own the fantastic results obtained when Neural Designer is used.

The following table summarizes the technical features of these tools that might impact their GPU performance.

|  | **TensorFlow** | **Neural Designer** |
| --- | --- | --- |
| **Written in** | C++, CUDA, Python | C++, CUDA |
| **Interface** | Python | Graphical User Interface |
| **Differentiation** | Automatic | Analytical |

The above table shows that TensorFlow is programmed in C++ and Python, whereas Neural Designer is entirely programmed in C++.

Interpreted languages like Python have advantages over compiled languages like C ++, such as their ease of use.

However, the performance of Python is generally lower than that of C++. Indeed, Python takes significant time to interpret sentences during the program’s execution.

On the other hand, TensorFlow uses automatic differentiation, while Neural Designer uses analytical differentiation.

As before, automatic differentiation has some advantages over analytical differentiation. Indeed, it simplifies obtaining the gradient for new architectures or loss indices.

However, the performance of automatic differentiation is, in general, lower than that of analytical differentiation:  
The first derives the gradient during the program’s execution, while the second has that formula pre-calculated.

Next, we use TensorFlow and Neural Designer to measure the energy consumption for a benchmark problem on a reference computer. The results produced by these platforms are then compared.

## **Benchmark application**

The first step is to choose a benchmark application that is general enough to conclude the performance of the machine learning platforms. As previously stated, we will train a neural network that approximates a set of input-target samples.

In this regard, an approximation application is defined by a data set, a neural network, and an associated training strategy.  
The following table uniquely defines these three components.

|  |  |
| --- | --- |
| **Data set** | * Benchmark: Rosenbrock * Inputs number: 1000 * Targets number: 1 * Samples number: 1000000 * File size: 22 GB ([download](https://www.neuraldesigner.com/files/datasets/R_new.rar)) |
| **Neural network** | * Layers number: 2 * Layer 1:   + -Type: Perceptron (Dense)   + -Inputs number: 1000   + -Neurons number: 1000   + -Activation function: Hyperbolic tangent (tanh) * Layer 2:   + -Type: Perceptron (Dense)   + -Inputs number: 1000   + -Neurons number: 1   + -Activation function: Linear * Initialization: Random uniform [-1,1] |
| **Training strategy** | * Loss index:   + -Error: Mean Squared Error (MSE)   + -Regularization: None * Optimization algorithm:   + -Algorithm: Adaptive Moment Estimation (Adam)   + -Batch size: 1000   + -Maximum epochs: 20000 |

Once the TensorFlow and Neural Designer applications have been created, we must run them.

## **Reference computer**

The next step is to choose the computer in which the neural network will be trained with TensorFlow and Neural Designer.  
The table below shows the features of the computer used for this instance.

|  |  |
| --- | --- |
| **Operating system:** | Windows 11 Home 64-bit |
| **Processor:** | Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 3192 Mhz, 6 Core(s), 12 Logical Processor(s) |
| **Physical RAM:** | 31.9 GB |
| **Device (GPU):** | NVIDIA GeForce GTX 1050 Ti |

Once the computer has been selected, we install TensorFlow (2.1.0) and Neural Designer (5.9.9) on it.

Below, the TensorFlow code used is shown.

**import** **tensorflow** **as** **tf**

**import** **pandas** **as** **pd**

**import** **time**

**import** **numpy** **as** **np**

**from** **tensorflow.keras.utils** **import** Sequence

#read data float32

filename = "C:/Users/Usuario/Downloads/rosenbrock.csv"

df\_test = pd.read\_csv(filename, nrows=**100**)

float\_cols = [c **for** c **in** df\_test **if** df\_test[c].dtype == "float64"]

float32\_cols = {c: np.float32 **for** c **in** float\_cols}

data = pd.read\_csv(filename, engine='c', dtype=float32\_cols)

x = data.iloc[:,:-**1**].values

y = data.iloc[:,[-**1**]].values

initializer = tf.keras.initializers.RandomUniform(minval=-**1.**, maxval=**1.**)

#build model

model = tf.keras.models.Sequential([tf.keras.layers.Dense(**1000**,activation = 'tanh', kernel\_initializer = initializer, bias\_initializer=initializer),

tf.keras.layers.Dense(**1**, activation = 'linear', kernel\_initializer = initializer, bias\_initializer=initializer)])

#compile model

model.compile(optimizer='adam', loss = 'mean\_squared\_error')

#train model

**class** **DataGenerator**(Sequence):

**def** **\_\_init\_\_**(self, x\_set, y\_set, batch\_size):

self.x, self.y = x\_set, y\_set

self.batch\_size = batch\_size

**def** **\_\_len\_\_**(self):

**return** int(np.ceil(len(self.x) / float(self.batch\_size)))

**def** **\_\_getitem\_\_**(self, idx):

batch\_x = self.x[idx \* self.batch\_size:(idx + **1**) \* self.batch\_size]

batch\_y = self.y[idx \* self.batch\_size:(idx + **1**) \* self.batch\_size]

**return** batch\_x, batch\_y

train\_gen = DataGenerator(x, y, **1000**)

start\_time = time.time()

**with** tf.device('/gpu:0'):

history = model.fit(train\_gen, epochs=**20000**)

**print**("Training time: ", round(time.time() - start\_time), " seconds")

## **Reference electricity consumption meter**

This section describes the device used for the energy consumption measurements so that the reader can reproduce the results obtained in the following section with maximum accuracy.

|  |  |
| --- | --- |
| **Device model** | Perel E305EM5-G |

## **Results**

The last step is to run the benchmark application on the selected machine with TensorFlow and Neural Designer and compare the energy consumed by those platforms during training.

The following figure shows the training time with TensorFlow.

As we can see, TensorFlow takes 30:14:30 to train the neural network for 20000 epochs (5.44 seconds/epoch).  
The final mean squared error is 0.0003. The overall energy consumption of the training process is 4.5 kWh, as shown below.

Finally, the following figure shows the training time with Neural Designer.

Neural Designer takes 21:03:43 to train the neural network for 20000 epochs (3.79 seconds/epoch). During that time, it reaches a mean squared error of 0.023. The overall energy consumption of the training process is 2.6 kWh, as shown below.

The following table summarizes the the most important metrics the two machine learning platforms yielded.

|  | **TensorFlow** | **Neural Designer** |
| --- | --- | --- |
| **Training time** | 30:14:30 | 21:03:43 |
| **Epoch time** | 5.44 seconds/epoch | 3.79 seconds/epoch |
| **Training speed** | 183,824 samples/second | 263.852 samples/second |
| **Total energy consumed** | 4.5kWh | 2.6 kWh |

The following chart depicts the energy consumed using TensorFlow and Neural Designer graphically in this case.

**Electric energy consumption**

TensorFlow

4.50 kWh

4.50 kWh

Neural Designer

2.60 kWh

2.60 kWh

As we can see, the energy consumption of Neural Designer for this application is **42** % lower than that of TensorFlow.

.

**Energy consumption of model view controller**

The rest of this article is structured as follows: Sect. [2](https://link.springer.com/article/10.1007/s11227-023-05202-6#Sec2) examines related work about green metrics and estimation models. In Sect. [3](https://link.springer.com/article/10.1007/s11227-023-05202-6#Sec6), we describe the process used to construct the MVC-CCsEM based on the analysis of 52 applications. Section [4](https://link.springer.com/article/10.1007/s11227-023-05202-6#Sec21) presents a study of twelve applications in which the MVC-CCsEM energy consumption model is used to estimate energy consumption. The threats to the validity of this work are discussed in Sect. [5](https://link.springer.com/article/10.1007/s11227-023-05202-6#Sec22). Finally, Sect. [6](https://link.springer.com/article/10.1007/s11227-023-05202-6#Sec23)﻿ concludes the paper and recommends directions for future research.

## 2 Related work

Both industry and academia have obtained successful results in constructing sustainable hardware and communications by reducing their power consumption [[9](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR9), [10](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR10)]. Because ICT solutions comprise both hardware and software, not only must the hardware be sustainability-aware, but the software must also reduce its power consumption in order to maximise the sustainability of ICT solutions [[11](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR11)]. The evaluation of ecological software depends on the application’s structure and the hardware infrastructure used for its deployment [[17](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR17)]. The energy consumed by an application cannot be fully isolated from the basal consumption of the hardware by which it is executed. It is possible to measure hardware without executing an application and then measure the application without measuring peripheral devices and other background processes. Hence, measuring the specific energy consumption of an application requires a calculation, which is in fact an estimation. Therefore, green metrics are considered estimations instead of measurements. Green metrics and models are used to measure different applications or versions of the same application that are executed on the same hardware to determine the greenest software solution.

### 2.1 Green metrics: estimation from software execution

Green metrics estimate the energy consumption of applications when they are being executed. Several metrics described in the literature are reviewed in this section.

To measure the energy consumption and hardware utilisation of software, Guldner et al. [[26](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR26)] defined a method for evaluating “energy consumption” and “processor utilisation”. To collect measurements using green metrics, Michanan et al. [[27](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR27)] developed an interface based on a collection of classes using a dynamic data structure called GreenC5 to evaluate applications that used different workloads.

The collection of green metrics is supported by tools that are used to implement dynamic analysis techniques by applying calculation models to obtain only the specific power consumption of an application. Some well-known tools that measure energy consumption are RAPL [[28](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR28)], Microsoft Joulemeter [[29](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR29)], jRAPL [[30](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR30)], PowerAPI [[31](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR31)], and Jalen [[32](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR32)]. In this work, we used Microsoft Joulemeter to measure the energy consumption of the applications. Sehgal et al. [[33](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR33)] also used this tool for experimentation in their research.

Lago et al. [[34](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR34)] collected 66 software metrics and models to measure and estimate energy consumption, 17 of which were related to software architecture [[20](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR20)]. Other works have used these metrics and tools to estimate the energy consumption of a specific kind of software architecture, such as product line architectures [[35](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR35)] or cloud software architectures. Specifically, sustainable cloud software architectures have several open challenges [[36](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR36)], previous studies have attempted to define algorithms and energy-aware techniques [[37](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR37), [38](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR38)] that use green metrics to support sustainable solutions.

On the other hand, some previous works have used green metrics to demonstrate the relationship between the quality of a code and power consumption. Cairo et al. [[39](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR39)] described how smells and cyclomatic complexity influence the appearance of errors in applications. Sehgal et al. [[33](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR33)] evaluated the impact of refactoring smells on total energy consumption. These previous works revealed that Cyclomatic Complexity and smells are key factors in energy consumption.

These works were based on green metrics, which required calculations to execute the software application and generate an energy consumption rebound [[18](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR18)]. In this work, we go beyond these previous studies by developing a solution to avoid the rebound power consumption of green metrics.

### 2.2 Green models: estimation from green metrics

The measurements of green metrics obtained from application execution allow for the construction of energy consumption estimation models, which are also called metrics, based on them and other kinds of metrics.

Chatzigeorgiou and Stephanides [[17](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR17)] proposed three metrics to estimate the energy consumption of software: executed instruction count measure (EIC), memory access count measure (MAC) and software energy (SEM). These metrics use external meter instruments to measure in runtime. Specifically, EIC measures the number of instructions of an application executed by the processor. MAC measures the number of memory accesses of an instruction, and SEM calculates the average energy cost of executing an instruction. Dufour et al. [[40](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR40)] proposed three “hot spot” metrics based on the methods and classes of an application: total execution frequency (TEF) of a method, class invoked frequency (CIF), and class invoked time (CITC). Energy wasting rate [[41](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR41)] is another metric that uses measurements during the execution, which are part of the software’s logic structure (number of classes, methods, and structures for data exchange). The values obtained from the metric allow the programmer to know which Java classes or methods need to be changed or refactored to save power. These previous works revealed that the size of an application (source lines of code (SLOC), number of classes, methods, etc.) is a key factor in its energy consumption. Therefore, size is part of our estimation model. However, these models are focused on size and do not estimate considering the quality of code.

In this regard, Fu et al. [[42](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR42)] estimated the power consumed by an application using statistical algorithms and machine learning. However, although this work was based on quality attributes, it had the disadvantage that the application must be executed to obtain the estimation, generating a rebound energy consumption effect. Because our goal was also to reduce the power consumption of our solution, we avoided this negative effect by defining an estimation model based on Size and quality attributes without the need to execute the application.

Regarding design patterns, Feitosa et al. [[43](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR43)] confirmed that design decisions also affect energy consumption. The authors used crossover experiments to estimate the energy consumption of solutions that applied the GoF design patterns state/strategy and template method. To validate the estimated energy measurements, they applied statistical models and the agglomerative hierarchical clustering technique using SLOC and message passing coupling (MPC) as metrics. This work was innovative in early work on estimations because it was based on design patterns and considered the role of architectural components, not only based on Size (SLOC). They also considered that a quality attribute was coupling (MPC). However, it is important not only to use one quality metric but also to use one that previous studies have revealed is a key factor in power consumption, such as Cyclomatic Complexity and smells [[33](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR33)].

Several previous studies have addressed the energy consumption estimation of software architectures. Regarding software architectures, Seo et al. [[20](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR20)] acknowledged the need to create specialised early estimation models that would enable different architectural patterns and styles to be precise. This capability also enables an engineer to employ energy cost predictions to determine the most appropriate architectural style for a given distributed application before the implementation of the system. This provides the opportunity to compare (i) the power consumption of different architectural patterns or architectural styles of the same application and (ii) the power consumption of different applications designed with the same architectural pattern or architectural style. In this work, Seo et al. [[20](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR20)] presented 17 architectural consumption metrics that estimated the energy consumption of components and connectors. They showed that, because components and connectors play diverse roles in architecture, they incur different wastages of energy consumption. Among the 17 metrics, it is important to emphasise the generic energy cost model because it measures a complete architecture by distinguishing the energy consumed by the components (i.e. computational elements) and connectors (i.e. interaction elements). The generic energy cost model specialises in other metrics, such as the client–server energy cost model and the pub–sub energy cost model, which facilitate the early energy estimation of architectural patterns. However, these estimation metrics are based on Size, and quality attributes are not included in the estimation.

### 2.3 Green models for estimating power consumption during software life cycle

Fast time-to-market and the wide adoption of agile methods have led software engineers to apply a fast design decision-making process that may result in sub-optimal sustainable decisions for software applications, if suitable sustainable tools are not adopted as part of the process. Ardito et al. [[44](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR44)] provided guidelines for correctly measuring software applications and describing the techniques, tools and models that can be used, whereas Georgiou et al. [[45](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR45)] drew attention to this problem throughout the entire software life cycle. They provided guidance for using tools and techniques during the requirements analysis, design, implementation, testing and maintenance of software applications. Ournazi et al. [[46](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR46)] also emphasised this problem and provided a solution for addressing sustainability in software requirements and measuring power consumption during software construction to fulfil green requirements. However, these tools are based on green metrics that require software execution, which leads to an undesirable power consumption rebound effect throughout the entire software life cycle.

## In summary, based on the relevant literature, it may be concluded that there are no early green models that do not require the execution of a software application to estimate its energy consumption by considering its design and quality attributes. To develop an integrated solution that takes into account the needs identified in our review of related studies (see Table [1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Tab1)), in this work an estimation model was developed to determine the power consumption of MVC software architectures based on not only size but also quality attributes, such as complexity and smells (i.e. code smells and/or Duplicated Lines)﻿. In addition, this early estimation model can be applied as often as necessary during software construction and maintenance processes without generating rebound power consumption effects. Finally, this early estimation model was specialised in the MVC pattern to facilitate comparison of the power consumption of different applications or versions Construction of the MVC-CCsEM model

To construct the MVC-CCsEM, we defined and followed a rigorous process formalised in the standard of Software and Systems Process Engineering Meta-Model (SPEM) [[47](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR47)] to guarantee its replication and reuse for creating new estimation models based on quality and energy metrics (see Fig. [1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig1)). The process of constructing the MVC-CCsEM comprised five phases: scope definition, building, profiling, analysis and construction. Each phase included a set of tasks as well as their inputs and outputs. Tasks related to energy estimation metrics required highly rigorous testing to avoid bias and obtain precise results. Therefore, the process was based on the evaluation of energy efficiency by Mancebo et al. [[48](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR48)] to define energy consumption measurement tasks. Each phase, its tasks and its results are described in the following subsections.

### 3.1 Phase I: scope definition

This phase consists of two tasks: specifying the requirements and the goal of the model. The requirements specification consists in defining the scope and context of the model, as well as the inclusion and exclusion criteria. The goal of the model is defined as a set of hypotheses and/or research questions (see Fig. [1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig1)). The “scope and context” input and the “inclusion and exclusion criteria”, “hypothetical model” and “hypotheses” outcomes are described in the following subsections.

#### 3.1.1 Scope and context

In our previous work, we defined the CCsEM model to estimate the energy consumption of software components without being executed (see Eqs. [1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Equ1) and  [2](https://link.springer.com/article/10.1007/s11227-023-05202-6#Equ2)) [[49](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR49)]. Equation [1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Equ1) presents the final model in terms of variables that represent size (SIZE and SLOC), complexity (CC) and maintainability (CS), and the model error (e) (see the variables description, Sect. [3.1.3](https://link.springer.com/article/10.1007/s11227-023-05202-6#Sec10)). However, it is important to consider that the obtained results are normalised by a logarithmic transformation. Therefore, to obtain the estimation value without normalisation, it is necessary to apply the inverse function of the log (CCsEM), that is, the exponential function. Therefore, the power consumption estimation of the model CCsEM is obtained by applying the exponential function, as shown in Eq. [2](https://link.springer.com/article/10.1007/s11227-023-05202-6#Equ2).

log(CCsEM)=4.187−0.5925log(SIZE)−0.84031log(SLOC)+0.3332log(CC)+0.4084log(CS)+�

(1)

CCsEM=exp(log(CCsEM))

(2)

This CCsEM model is based on the metric Generic Energy Cost Model [[20](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR20)], which estimates the energy consumption of software architectures in terms of the energy costs of components and connectors. The CCsEM model was narrowed to facilitate the energy consumption estimation of components and simple interactions to avoid the uncertainty generated by complex connectors. They vary from simple interactions [[50](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR50)] between components to complex orchestrators among components that implement coordination protocols [[51](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR51)]. In fact, the energy consumption of complex connectors that orchestrate a high number of components with complex policies may have a high impact on energy consumption even higher than the components.

Complex connectors are applied by a wide variety of architectural patterns [[23](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR23)] and technologies [[52](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR52),[53](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR53),[54](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR54)] which influence energy consumption to different degrees. Therefore, they require a separate specific study to define an appropriate estimation model and determine their energy consumption, depending on their role, technology, protocol and number of orchestrated components, among other properties. In the present work, we went beyond the CCsEM model by considering both components and connectors, that is, the complete architecture. However, because the complex connector variability and their derived uncertainty must be avoided, this work was constrained to a specific architectural pattern, following the recommendations of Seo et al. [[20](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR20)] to construct one power consumption estimation model for each specific architectural pattern or style. In this work, the CCsEM model is specific to software architectures that implement the MVC pattern. Therefore, the training of the model is also constrained by the execution of applications that implement MVC software architectures.

Guaman et. al [[55](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR55)] revealed the metrics that are the most frequently used as static analysis tools to examine software architectures and applications. In addition to size, these metrics include complexity and maintainability. Maintainability is usually supported by tools that provide technical debt (TD) analysis. These TD tools measure smells, such as code smells and duplicated lines. In addition, Sehgal et al. [[33](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR33)] found that cyclomatic complexity and smells are vital factors in power consumption. This base of knowledge was used to construct a quality-aware energy estimation model to predict energy consumption before execution. In this work, the metrics that were measured to construct the MVC-CCsEM are as follows:

* Size: Size was measured by the application Size ﻿and SLOC.
* Complexity: Complexity was measured by Cyclomatic Complexity.﻿
* Maintainability: Maintainability was measured by smells, Duplicated Lines ﻿and code smells.

These decisions allowed the model to fulfil the needs identified in the software power consumption estimation field (see Table [1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Tab1)). In addition, it is important to emphasize the application context of the model. The MVC-CCsEM model was defined to support a sustainable software construction and maintenance without the need for execution. This means that the model can be calculated as many times as the engineer requires without incurring additional power consumption. This calculation during the software construction and maintenance is locally performed by an engineer on a computer; obtaining the measurements of the quality and size from tools such as SonarQube [[56](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR56), [57](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR57)]. Therefore, the model is independent of the computer or IoT device on which it is deployed and an MVC-CCsEM-improved greener code during software development fosters greener ICT solutions if we deployed it on power-efficient computers and IoT devices in the real setting. Then, these greener ICT solutions can be additionally measured using green metrics and models (see Sects.  [2.1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Sec3) and  [2.2](https://link.springer.com/article/10.1007/s11227-023-05202-6#Sec4)) in their real setting to measure different devices, communications and user-connections.

(3)

*Let be* �0: *the value of the dependent variable when the rest of the variables are set to zero;* ��: *the regression coefficient of its independent variable, that is, the average effect of a unit increment of the independent variable on the dependent variable, being i=SIZE, SLOC, CC, CS or DL; and e: the model error.*

The *H2* is validated during the construction of the model. If the two variables of maintainability are not significant or the CC is not significant, the hypothesis *H2* is rejected. If the *H2* is accepted, then the hypothesis *H1* is validated with the model execution by determining if the estimation results of MVC-CCsEM are accurate for its adoption. To address our goal both hypotheses *H1* and *H2* must be fulfilled, which means the rejection of *H0*.

### 3.2 Phase II: building

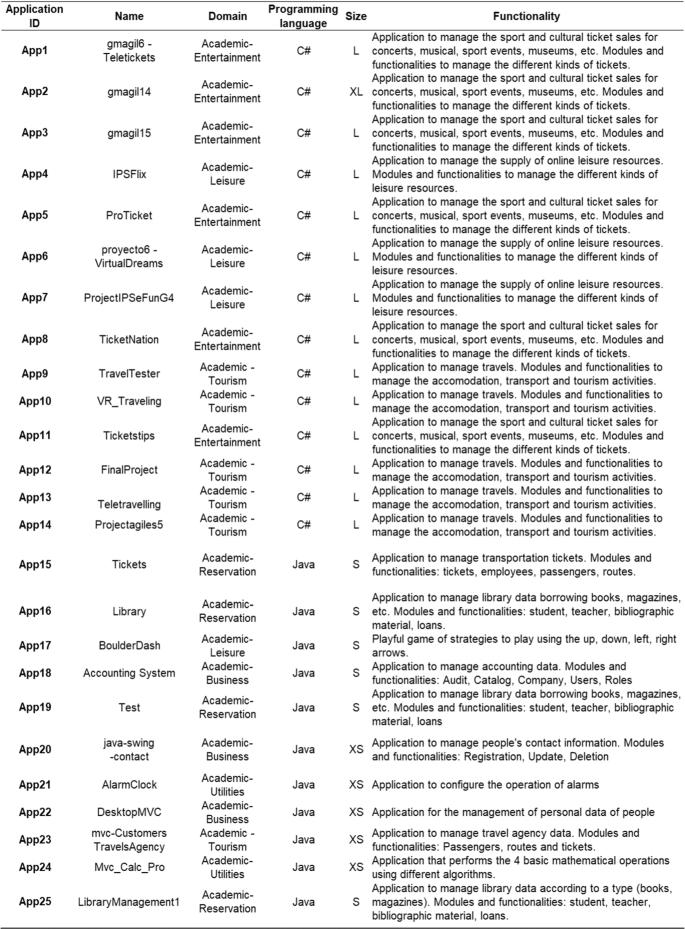
To construct the model, a training corpus data set is required. It must be composed of a set of MVC applications and their quality and energy measurements. This data set is created in the building phase. It consisted of two tasks: “SW architecture search and selection” and “Build applications” (see Fig. [1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig1)). The tasks are described in the following subsections.

#### 3.2.1 SW architecture search and selection

This first task aims to search the training corpus of MVC applications in public and accessible repositories. By taking into account the inclusion criteria (IC) and exclusion criteria (EC), a search of the C# and Java MVC applications was performed in GitHub (<https://github.com/>), where the software code is accessible and was written by different programmers to avoid human and programming bias. The applications were then downloaded from GitHub and stored locally on the computer. In this search, we obtained 74 applications. The second step consisted of confirming that none of the downloaded applications fulfilled EC1. Therefore, the applications were opened in their corresponding integrated development environments (IDE), and it was confirmed that their architecture implemented an MVC pattern. When it was confirmed that our data set was composed of MVC applications, it was possible to address the second task. At this point, we excluded 19 applications that applied EC1, and 55 applications were preserved for the data set.

#### 3.2.2 Build applications

This task requires two actions (1) compile and execute the MVC applications and (2) determine that they did not fulfil EC2 and EC3. To that end, the applications must be compiled and executed using the corresponding Integrated Development Environment (IDE). This execution could include pre-processing the code, such as updating dependencies or preparing the JAR files for Java applications. In this task, three applications were removed from the data set by applying EC2, EC3 and EC4. The resulting data set of 52 MVC applications was obtained (see Tables [2](https://link.springer.com/article/10.1007/s11227-023-05202-6#Tab2) and [3](https://link.springer.com/article/10.1007/s11227-023-05202-6#Tab3)). These applications encompass a range of sizes in percentages of 36.5 % XS, 34.6% S, 1.9 % M, 25% L, and 1.9%XL. Specifically, 73.1% are codified in Java, and 26.9% are codified in C#, and each addresses different domains, as shown in Table [2](https://link.springer.com/article/10.1007/s11227-023-05202-6#Tab2) and Fig. [2](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig2).

**2**

### Phase III: profiling

The Profiling phase consists of two tasks: “Static Analysis” and “Dynamic Analysis” (see Fig. [1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig1)). The Static Analysis is conducted to calculate the size, complexity and maintainability metrics from the selected applications. The Dynamic Analysis is conducted to measure the energy consumption of the applications during their execution. These measurements constitute the training corpus of the model. These two tasks are described in the following subsections.

#### 3.3.1 Static analysis

The static analysis was performed using the SonarQube tool [[57](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR57)]. Each application was loaded in SonarQube, and its SIZE, SLOC, CC, DL and CS were collected by executing the sonar-scanner command. The collected data were then stored in a MySQL repository by SonarQube. We also stored the data in an Excel file for processing.

#### 3.3.2 Dynamic analysis

The dynamic analysis was performed using Microsoft Joulemeter, which measures the energy consumed by the execution of applications and stores the measurements in a CSV file. Specifically, the measurements provided by Microsoft Joulemeter are CPU, MONITOR, RAM, BASE, Application Power (CPU only), and Total Power (see Fig. [3](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig3)). To obtain valuable data from the dynamic analysis, it is required that the context of execution and the collection procedure be the same for all the applications under measurement [[48](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR48)]. Because all the applications are measured under the same execution conditions, Joulemeter based their calculation on the same basal consumption, which guaranteed that we compare only the power consumption of the application, and the measurement is independent of the computer. Next, we define the context of execution and the collection procedure followed during the dynamic analysis:

* *Context of Execution*: It is important to include both hardware and software configurations.
  + *Hardware*: All applications were executed on a laptop with the same hardware configuration: Processor: ADM Ryzen 5 4600 H with Radeon Graphics 3.GHz, RAM: 16.0 GB.
  + *Software*: All applications were deployed on the same software stack and configured with the same parameters. The operating system, the Integrated Development Environment (IDE), and the database for executing the Java and C# applications are the following: Operating System: Windows 11 Home 64 bits. IDE: NetBeans 8.6 (Java) and Microsoft Visual Studio Enterprise 2019 (C#), DBMS: MySQL 5.6 (Java) and SQL Server (C#).
* *Collection procedure*: This is composed of several steps, which are detailed as follows:
  + 1.

*Calibrating*: Microsoft Joulemeter was calibrated running on batteries with the defined software stack (power chord disconnected). Subsequently, it was not possible to update the hardware and software established in the context of execution until the collection procedure was finished.

* + 2.

*Determining the measuring time of applications*: During the execution of an application, energy consumption fluctuated, depending on the functionality that was being executed at that moment. Hence, to ensure that the measurement of an application is representative, the execution time must allow the execution of all its functionalities. To determine this time systematically, we defined the following steps:

* + 1. (a)

The applications categorized with the two highest SIZE of the training corpus are selected in order to determine the number of functionalities provided for execution. In this case, we selected applications that were categorised as XL and L, and we calculated the number of functionalities. For example, if the application provided the create-read-update-delete (CRUD) of a specific element of the software system, it was recorded as having four functionalities.

* + 1. (b)

Applications with a high number of functionalities in the previous step are selected. We selected all applications that had the three highest numbers of functionalities.

* + 1. (c)

The applications selected in the previous step are run by executing all their functionalities and timing the duration of the entire execution. We run the applications and their execution times were recorded.

* + 1. (d)

From the times extracted in the previous step, the highest time is selected and rounded to minutes. In our case, the time was 10 min.

* + 3.

*Establishing the execution context*: The unnecessary peripheral devices and background processes were disconnected and stopped, respectively. Only Microsoft Joulemeter and the software required to execute the application were left running. In addition, to begin the measurement at the same starting point in applications that managed databases, all databases were emptied beforehand.

* + 4.

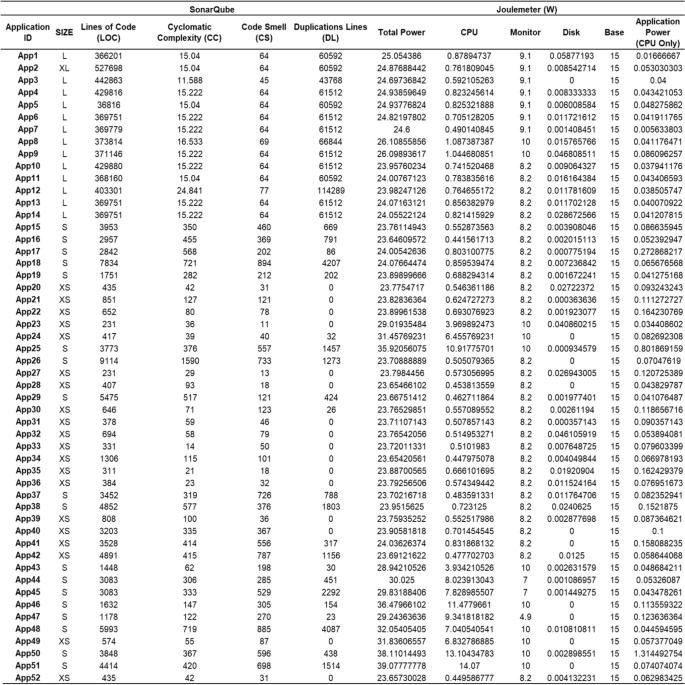
*Measuring power consumption*: Each application was measured during the time determined in step 2, which was 10 min. It is recommended to follow the same interaction pattern to avoid neglecting any functionality during execution. Small applications in which the execution of all functionalities was finished before the established time were executed until the time was completed by repeating the interaction pattern as many times as necessary.

* + 5.

*Storing power consumption measurements*: After completing step 4 in each application, the obtained measurements were stored.

Steps 4 and 5 were performed iteratively. When they were executed for all applications, the training corpus of the model was obtained, which in this case was composed of 52 MVC applications (see Fig. [3](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig3)).

**Fig. 3**



#### 3.4.1 Data analysis

In this task, we prepared and selected the energy consumption values provided by Microsoft Joulemeter and used them to train the regression model (see Fig. [4](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig4)). Microsoft Joulemeter calculates power consumption using its estimation mathematical model, which is based on CPU, screen, memory and storage of total power [[58](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR58)]. Total power and CPU metrics cannot be used as estimation variables because they include measurements not only from the execution of the application under study but also from other processes that are being executed at the same time. However, in Joulemeter, the power consumption estimation model also calculates the “application power (CPU only)”, that is, it subtracts the total CPU and the basal CPU consumption of the computer (see Eq. [4](https://link.springer.com/article/10.1007/s11227-023-05202-6#Equ4)). Hence, the representative metric is “application power (CPU only)” because it stores the energy consumption of the MVC application during its execution-that is, the strict power consumption of the applications under evaluation. In addition, the “data analysis” task (see Fig. [1](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig1)) requires analysing the relationship between the variables to identify two possible problems: (**P1**) there are independent variables that present non-linear relationships with the dependent variable, and (**P2**) there are collinearity problems between the independent variables.

ApplicationPower(CPUonly)=CPU−BasalCPUConsumption

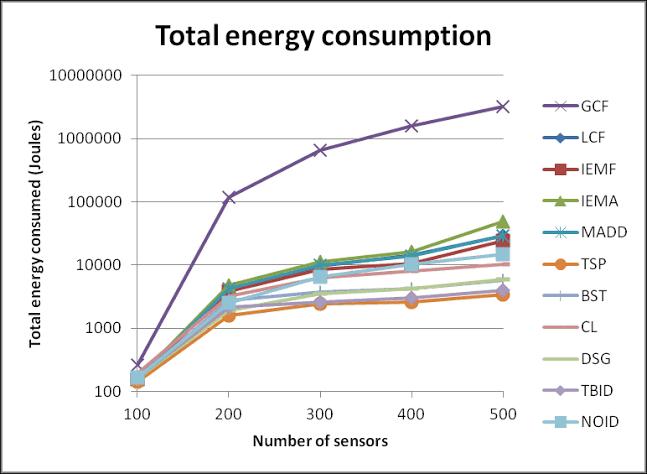
(4)

To analyse the data distribution, we used the GGally package [[59](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR59)], which is able to plot a data set with multiple variables. Figure [2](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig2) shows the GGally results of the data set in histograms and scatter plots. They show a linear relationship among the variables because the values of the scatter plots between two independent variables are close to the line; thus, problem **P1** regarding non-linear relationships was avoided. The data distribution showed that some variables were linearly dependent

## Estimating the energy consumption of MVC applications using the MVC-CCsEM

Once the MVC-CCsEM model was constructed and *H2* was validated (see Phase I), it was necessary to validate *H1* by evaluating the model’s estimation capabilities and accuracy. To that end, an experiment was conducted in which the energy consumed by twelve MVC applications was estimated using the model MVC-CCsEM, and it was compared with the power consumption value measured by Joulemeter during the execution of the MVC application (i.e. application power (CPU only)). The characteristics of these twelve applications are described in detail in Table [5](https://link.springer.com/article/10.1007/s11227-023-05202-6#Tab5). To conduct this experiment, the Profiling phase was performed for all twelve applications according to the construction process of the model. The metrics SIZE, SLOC, CC and DL were collected by the SonarQube tool to determine the values of the independent variables of the MVC-CCsME model. In addition, to collect data on the real consumption of the MVC applications, they were executed in the same context and pattern to extract the application power (CPU only) using Microsoft Joulemeter. The results are presented in Table [5](https://link.springer.com/article/10.1007/s11227-023-05202-6#Tab5).

Data set: description of applications

Figure [5](https://link.springer.com/article/10.1007/s11227-023-05202-6#Fig5) and Table [5](https://link.springer.com/article/10.1007/s11227-023-05202-6#Tab5) show the comparison in joules between real energy consumption and the estimation of the MVC-CCsEM model. To measure the model’s performance and determine the precision with which the model predicted the dependent variable, the root mean squared error (RMSE) was used [[66](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR66)], which was calculated using R-Studio. In this case, the prediction of these twelve applications was RMSE = 0.2861, which was an acceptable value for the estimation [[66](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR66),[67](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR67),[68](https://link.springer.com/article/10.1007/s11227-023-05202-6#ref-CR68)]. The estimation of Application 2 was the most accurate, with a difference of �=0.0041, whereas Application 5 had the highest error of estimation, �=0.0780 (see Table [5](https://link.springer.com/article/10.1007/s11227-023-05202-6#Tab5)). Based on the results of this analysis, we concluded that the estimation of the model was sufficiently accurate to be a valuable tool for software engineers. The joule estimation was very close to the reality in most of the applications. Thus, the need to execute the application to determine its power consumption and generate extra consumption was avoided. Based on these results, the MVC-CCsEM model could effectively support software engineers during software construction and maintenance by informing them about variations in power consumption due to the code changes. Software engineers can therefore avoid performing explicit measurements, such as new tasks in the software life cycle, executing an application to obtain information and therefore generatin

## Conclusion

In this work, we developed a green estimation model, MVC-CCsEM, which provides an integrated solution for the identified needs of software power consumption estimation throughout the software lifecycle. MVC-CCsEM estimates the energy consumption of MVC applications in terms of Size, Source Lines Of Code, Cyclomatic Complexity and Duplicated Lines ﻿without the need for execution and the resulting generation of power consumption rebound effects. The adoption of this model is feasible during the construction and maintenance decision-making of C# and Java applications of any size in which the architecture implements an MVC pattern that uses MySQL and SQL server databases. The model was constructed by analysing 52 applications and validated by estimating twelve applications. The results showed that this model is an advancement in the field. The results for the twelve applications showed an RMSE = 0.2861, indicating that the joule estimation was very close to reality in avoiding the extra energy consumed by application execution. Therefore, MVC-CCsEM can assist software engineers saving their time during their development tasks and the power consumption of their applications.

In addition, this work formalises in SPEM the process that was defined and followed to construct the MVC-CCsEM model, which is a reusable asset for the research community, by emphasising optional and mandatory tasks as well as defining how to perform them to avoid bias. In future work, we will extend and improve the MVC-CCsEM model by extending the training corpus with a greater number of applications, examining the ways in which a population and the use of MVC scaffolding mechanisms may influence power consumption, and measuring each application several times to increase the accuracy of the measurements. In addition, we plan to automatise the calculation using Sonarqube to construct a tool. We will examine other variables that can be extracted from a static analysis to determine whether they influence the energy consumption of software applications. In addition, we will determine how new languages and databases influence the model to determine whether it is necessary to specialise in programming languages or data management systems. Finally, we will determine the success of adopting the model during the software life cycle for the benefit software engineers.